CSC411 Machine Learning
Project 3: Fake News

Ariel Kelman
Student No: 1000561368

Gideon Blinick
Student No: 999763000

19 March 2018

1 Dataset Description

Both datasets (real and fake) contain headlines about U.S. President Donald Trump. From the headlines, it is very much possible to determine with significant accuracy the percentage chance that a given headline is fake or real based on the precense of absence of certain words in the headline. For example, after performing some preliminary analyses on the headlines, we discovered that the following 3 words might be of particular use.

1. The word "donald" appears 42.05% of the time in real headlines while appearing in only 17.55% of fake headlines. This 24.5% difference was by far the largest of any word.

2. The word "the" appears 27.9% of the time in fake headlines while appearing in only 7.9% of real headlines for a difference of 20.02%.

3. The word "trumps" appears 11.1% of the time in real headlines while appearing in only 0.3% of fake headlines for a difference of 10.8%.

2 Naive Bayes Implementation

In part 2, we implemented the Naive Bayes algorithm on our training, validation and testing sets. To acheive this, we divided our solution into 3 functions and proceeded as follows:

1. In our first function "training_part" we use the training set to achieve estimates for p(real), p(fake), $p(word\|real)$ and $p(word\|fake)$. We estimated $p(word\|real)$ and $p(word\|fake)$ by counting the number of headlines in the training set where the word appears and the headline is real, added a factor of $m \cdot p$, and divided this result by the size of the training set plus m. We also included words that were not found in the training set (but found in validation or testing) and set their values to $\frac{m \cdot p}{count(headlines) + m}$. The implementation of our function is"

```
def training_part(fake_lines_training_set, real_lines_training_set, m, p):
    '''Takes in a list of training set lines divided into real and fake groups.
    Returns a list of probabilities that represent p(word|real) and p(word|fake).
    Therefore, this list will be 5833 long to account for every word.
    '''
```

```
#Part 1: Training: calculate p(real), p(fake), and p(word|real), p(word|fake)
fake_stats_training_set = get_stats(fake_lines_training_set) #compute
probabilities for each word
real_stats_training_set = get_stats(real_lines_training_set)

fake_counts_training_set = get_count(fake_lines_training_set) #compute counts
 for each word
real_counts_training_set = get_count(real_lines_training_set)

p_fake = len(fake_lines_training_set)/(len(fake_lines_training_set) + len(
real_lines_training_set))
p_real = len(real_lines_training_set)/(len(fake_lines_training_set) + len(
real_lines_training_set))

    # what we want is p(real|words in headline).
    # this is equal to p(words in headline|real)*p(real), divided by
    #   p(words in headline|real)*p(real) + p(words in headline|fake)*p(fake)
    # We already have p(real) and p(fake) from above, so we just need to find
    # p(words in headline|real) and p(words in headline|fake)
    # p(word in headline|real) is equal to count(word & real) + mp
    # divded by count(real) + m.
    # So we need to choose values for m and p and then come up with a
    # dictionaries of the adjusted probabilities.

all_words = list(real_counts_total.keys())

missing = { x:0 for x in fake_counts_total.keys() if x not in
real_counts_training_set.keys() }
real_counts_training_set.update( missing )
missing = { x:0 for x in real_counts_total.keys() if x not in
fake_counts_training_set.keys() }
fake_counts_training_set.update( missing )


adjusted_fake_counts_training_set = {}
adjusted_real_counts_training_set = {}



for word in all_words:
    adjusted_fake_counts_training_set[word] = fake_counts_training_set[word]
+ mp
    adjusted_real_counts_training_set[word] = real_counts_training_set[word]
+ mp

naive_divisor = len(all_words) + m

adjusted_fake_stats_training_set = {} #P(w | fake)
adjusted_real_stats_training_set = {} #P(w | real)

for word in all_words:
    adjusted_fake_stats_training_set[word] =
adjusted_fake_counts_training_set[word]/naive_divisor
    adjusted_real_stats_training_set[word] =
adjusted_real_counts_training_set[word]/naive_divisor


return p_fake, p_real, adjusted_fake_stats_training_set,
adjusted_real_stats_training_set
```

2. In the second part , we created a function called "evaluate" which uses the outputs of the first function - p(real), p(fake), $p(word \| real)$, $p(word \| fake)$ - as its inputs, uses them to compute

$p(headline\|real)$ and $p(healine\|fake)$ for a given set of headlines, and in turn uses those to compute $p(real\|headline)$ and $p(fake\|headline)$ for the set of headlines, which is the function output. In in this function that we utlize the fact that

$$a_1 \cdot a_2 ... = e^{loga_1 + loga_2 ...} \tag{1}$$

, in order to compute $p(headline\|real)$ as the product of $p(word\|real)$ for all words in and outside the statement. Performing the calculation in this way avoids underflow problems. The implementation of the function is:

```
def evaluate(p_fake, p_real, adjusted_fake_stats_training_set,
    adjusted_real_stats_training_set, SET):
    ''' Calculate p(fake|headline) for all headlines in a given set (TRAINING,
    TEST, VALIDATION).
    Takes p(real), p(fake), p(word|real), p(word|fake) as parameters.'''

    total_real_probabilities = {}
    total_fake_probabilities = {}

    all_words = list(real_counts_total.keys())

    for headline in SET:

        probabilities_real = []
        probabilities_fake = []

        headline_words = list(set( headline.split(' ') )) #converting to set and
    back to a list removes duplicates
        for word in headline_words:
            probabilities_real.append(adjusted_real_stats_training_set.get(word))
            probabilities_fake.append(adjusted_fake_stats_training_set.get(word))
        non_headline_words = [x for x in all_words if x not in headline_words]
        for word in non_headline_words:
            probabilities_real.append(1 - adjusted_real_stats_training_set.get(
    word))
            probabilities_fake.append(1 - adjusted_fake_stats_training_set.get(
    word))

        total_real_probability = 0
        total_fake_probability = 0
        for k in range(len(probabilities_real)):
            total_real_probability = total_real_probability + math.log(
    probabilities_real[k])
            total_fake_probability = total_fake_probability + math.log(
    probabilities_fake[k])
        total_real_probability = math.exp(total_real_probability)
        total_real_probabilities[headline] = total_real_probability
        total_fake_probability = math.exp(total_fake_probability)
        total_fake_probabilities[headline] = total_fake_probability

    ## At this point, we have calculated p(headline|real) and p(headline|fake)
    for all
    ## headlines in whatever set we are testing
    ## Now we need to find p(fake|headline) which means multiplying p(headline|
    fake) by p(fake) and
    ## dividing by p(headline|fake)*p(fake) + p(headline|real)*p(real)

    #Compute final probabilites
    numerator_fake = [p_fake*x for x in total_fake_probabilities.values()]
    numerator_real = [p_real*x for x in total_real_probabilities.values()]

    final_fake = [numerator_fake[i]/(numerator_fake[i] + numerator_real[i]) for i
     in range(len(numerator_fake))]
    final_real = [numerator_real[i]/(numerator_fake[i] + numerator_real[i]) for i
     in range(len(numerator_real))]
```

```
    return final_fake , final_real
```

3. Finally, in the third part, we have a function called "check_accuracy" to determine the percentage of statements correctly classified. It does this by accepting $p(fake\|headline)$ from the previous function, as well as an array indicating whether a headline is fake by a 1 or real by a 0. The function then creates a new array that accepts $p(fake\|headline)$ for a given headline and rounds it to 1 if it is 0.5 or greater, and down to 0 otherwise. The function then subtracts this new array from our checker array and counts the number of non-zero elements. These elements represent incorrect classifications and allow us to determine our accuracy. Our function implementation is:

```
def check_accuracy(final_fake, y):
    '''Given a list of fake probabilities, compare with the actual results by
    rounding.
    If the fake  probability is greater than 0.5, consider it fake and if less,
    consider it real.
    Output the accuracy rate'''
    ## At this point we are done and have found p(fake|headline).
    ## All that remains is accuracy checking
    ## This part takes final_fake and one of y_tr, y_va, y_te as parameters
    ## and returns a percentage value in accuracy

    pred_fake = np.array([round(x) for x in final_fake])

    y_2 = np.array(y)

    incorrect = np.count_nonzero(pred_fake - y_2)
    total = len(y)
    correct = total - incorrect
    accuracy = correct/total

    return accuracy
```

In choosing our parameters m and p for the model, we began with a value of m = $2 \cdot 5833$ and $\frac{1}{2 \cdot 5833}$ and optimized from there. This was chosen as our starting point because as mentioned on Piazza, it allows for 1 example per class per word and is intuitive. We then chose 4 different values of m and p and tested how they perform on our validation set, via a function called "optimize_mp". Our optimal m and p values were found to not matter so much,as long as they weren't $\frac{1}{1 \cdot 5833}$ and 5833, respectively. If we chose higher values for m and corresponding lower values for p, we achieved an acuracy on the validation set of 91.3%.

m	accuracy
5833	90.8%
$2 \cdot 5833$	91.3%
$3 \cdot 5833$	91.3%
$4 \cdot 5833$	91.3%

Our optimize_mp function is:

```
def optimize_mp(fake_lines_training_set, real_lines_training_set, m_s, mp):
    val_acc = {}
    for m in m_s:
        print(m)
```

```
    p_fake, p_real, adjusted_fake_stats_training_set,
adjusted_real_stats_training_set = training_part(fake_lines_training_set,
real_lines_training_set, m, mp)
    final_fake, final_real = evaluate(p_fake, p_real,
adjusted_fake_stats_training_set, adjusted_real_stats_training_set, validation_set
)
    val_acc[m] = check_accuracy(final_fake, y_va)
return val_acc
```

Finally, on the training set we obtained an accuracy of 92.5% while on the testing set we achieved an accuracy of 92.2%.

3 Predictive Factors

3.1 Words

The following 4 images present the top 10 words for each relevant probability. Note that for some classes, such as $p(real||word)$, there were many other words that could have been in the list provided. Also note that for $p(real||word)$ the probabilities are slightly greater than 1 due to rounding.

1. $p(real||word)$

p_reallword_top - List (10 elements)

Index	Type	Size	Value
0	tuple	2	(1.0000000000000002, 'whats')
1	tuple	2	(1.0000000000000002, 'visits')
2	tuple	2	(1.0000000000000002, 'use')
3	tuple	2	(1.0000000000000002, 'turnbull')
4	tuple	2	(1.0000000000000002, 'transgender')
5	tuple	2	(1.0000000000000002, 'transcript')
6	tuple	2	(1.0000000000000002, 'third')
7	tuple	2	(1.0000000000000002, 'saturday')
8	tuple	2	(1.0000000000000002, 'react')
9	tuple	2	(1.0000000000000002, 'parts')

2. $p(real||not\_word)$

p_reallnot_word_top - List (10 elements)

Index	Type	Size	Value
0	tuple	2	(0.930635838150289, 'trump')
1	tuple	2	(0.6621551275377409, 'the')
2	tuple	2	(0.6362599771949828, 'to')
3	tuple	2	(0.6305555555555556, 'hillary')
4	tuple	2	(0.626609442060086, 'a')
5	tuple	2	(0.6206735266604303, 'clinton')
6	tuple	2	(0.6203339882121808, 'for')
7	tuple	2	(0.6195601310248012, 'is')
8	tuple	2	(0.6193208990913439, 'and')
9	tuple	2	(0.6188068756319515, 'in')

3. $p(fake\|word)$

p_fakelword_top - List (10 elements)

Index	Type	Size	Value
0	tuple	2	(1.0000000000000002, 'won')
1	tuple	2	(1.0000000000000002, 'uses')
2	tuple	2	(1.0000000000000002, 'thug')
3	tuple	2	(1.0000000000000002, 'these')
4	tuple	2	(1.0000000000000002, 'taking')
5	tuple	2	(1.0000000000000002, 'switching')
6	tuple	2	(1.0000000000000002, 'swamp')
7	tuple	2	(1.0000000000000002, 'shocks')
8	tuple	2	(1.0000000000000002, 'roberts')
9	tuple	2	(1.0000000000000002, 'poll')

4. $p(fake\|not\_word)$

p_fakelnot_word_top - List (10 elements)

Index	Type	Size	Value
0	tuple	2	(0.48578811369509045, 'donald')
1	tuple	2	(0.42468324730173623, 'trumps')
2	tuple	2	(0.4234277484397504, 'us')
3	tuple	2	(0.4138257575757575, 'says')
4	tuple	2	(0.40853932584269664, 'north')
5	tuple	2	(0.4081724292770543, 'korea')
6	tuple	2	(0.40644583706356313, 'ban')
7	tuple	2	(0.4041796353935082, 'turnbull')
8	tuple	2	(0.4038205242114616, 'travel')
9	tuple	2	(0.40267857142857144, 'wall')

These values were obtained by applying Bayes rule. More specifically, we calculated $p(real\|word)$ by $\dfrac{p(word\|real) \cdot p(real)}{p(word)}$. $p(word\|real)$ can be obtained for every word by taking the number of headlines with a given word in the training set that are also real and dividing it by the total number of headlines that are real in the training set. $p(real)$ is the number of headlines in the training set that are real divided by the total number of headlines in the training set. Finally, p(word) is the number of headlines in the training set with the word divided by the number of headlines in the training set. All these are easily obtained.

To obtain $p(real\|not\_word)$, we use $\dfrac{p(not\_word\|real) \cdot p(real)}{p(not\_word)}$. p(real) is the same from the previous step, $p(not\_word\|real)$ is 1 - $p(real\|word)$, and $p(not\_word)$ is the number of headlines without a word in the training set divided by the total number of headlines. The calculcations for $p(real\|word)$ and $p(real\|not\_word)$ proceed in an equivalent way with an appropriate substitution of "fake" for any mention of "real".

What is observable from these results is that the precense of certain words has far greater predictive ability than the absence of any given word. This is logical, as certain words that appear rarely and only in either real or fake headlines will be entirely associated with the class they are found in. By contrast, the absence of any one word is not enough to definatively assign a class of real or fake since the remaining headlines without the given word do not all fall into one class or another.

4 Logistic Regression

5 Logistic Regression vs. Naive Bayes

When forming a prediction on a headline, both Naive Bayes and Logistic Regression effectively compute

$$\theta^T I = \theta_0 + \theta_1 I_1 + \theta_2 I_2 + ... + \theta_k I_k > thr$$

where thr is a given threshold value; in this project $thr = 0.5$. In both cases, the θ's represent the paremeters that define the prediction model, while the I's are a function of the input for which a prediction is produced.

For logistic regression, each $I = I(x)$ for a headline is simply a vector of 0's and 1's, where each position in the vector represents a word that was included in the training set. A 1 indicates that the word is found in the headline, while a 0 indicates that one was not. As the headlines are relatively short in comparison to the total number of words, this vector is very sparse. The θ's represent how important a feature (the presence of a particular word) is to the final prediction (θ_0 is a bias that can account for the priors on real vs. fake headlines).

For naive bayes,

6 Analysis of Logistic Regression

7 Decision Tree

7.1 Classification

Using the `sklearn` implementation of decision trees, we trained several decision trees to differentiate between the fake and real headlines. After some experimentation with the many parameters in the `sklearn DecisionTreeClassifier` (particularly with the maximum number of features used when looking for the best split), the default parameters gave the best results. As a split condition, maximum entropy was used rather than gini impurity, though similar results were obtained for both. Many of the settings served as ways to limit the size of the decision tree, so this result is not unexpected.

Decision trees with a maximum depth of $\{2, 3, 5, 10, 15, 20, 35, 50, 75, 100, None\}$ were built, and the results on the training, validation, and testing sets are shown in the figure below. The final point, not plotted, with no limit on the maximum depth, gave an accuracy of $1, 0.760.76$ on the training, validation, and testing sets respectively. As can be seen from the figure, the larger the depth of the decision tree, the greater accuracy achieved on the testing set. However, improvement is small on the validation and testing sets after a depth of 20. Without any other constraints (such as a minimum number of samples to split a node), a decision tree can reach perfect accuracy on the training set, as demonstrated above. Predictably, this leads to much greater preformance on the training set than on the validation and testing sets (though there is no *decrease* in preformance on the latter sets); showing the importance of using a validation and testing set to measure preformance.

7.2 Visualization

The following image shows the first few layers of the decision tree with depth 20. It was generated by saving a text representation of the tree (as a `.dot` file), which was then visualized by using the

It is interesting to look at the words being split on in these layers; X was a list of all words that appeared in either the fake or real datasets. These words are 'donald' (23), 'trumps' (1604; note the double appearance), 'the' (81), 'hillary' (44), and 'trump' (3). The exact same process of training was run ignoring stop words, but results were slightly worse; graphs similar to those mentioned above can be found in the `resources` directory (each filename indicates whether stop words were included, as

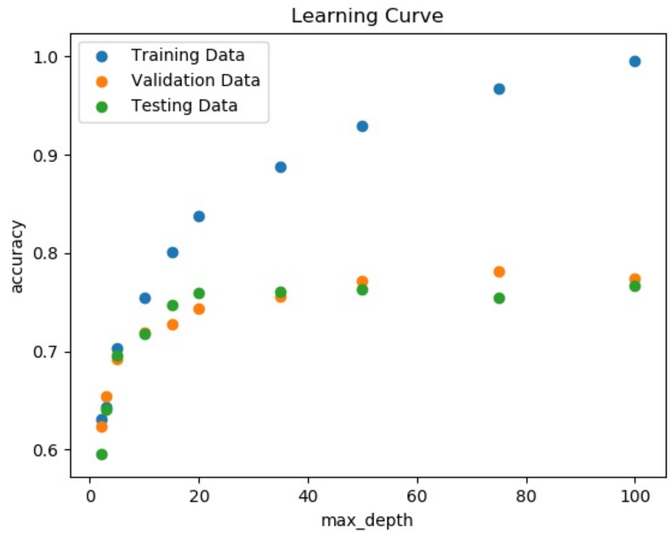

Figure 1: Plot showing the preformance of `sklearn` decision trees with varying depth. The split condition was maximum entropy, there was no limit on the maximum number of features for a split, and stop words were included.

Figure 2: The top few layers of the decision tree (depth = 20).

well as info on the other non-default parameters). Note that because of the use of dictionaries (i.e. because `dict.keys()` does not keep track of order), running the code again may result in different indices for each of the above words.

A visualization of the entire tree is saves as `part7b_all.pdf` in the `resources` directory; the text represenations of many trees that were generated during training are saved in `resources/part7`.

7.3 Comparison of All 3 Classifiers

All three classifiers preform much better than random guessing...

8 Information Theory

8.1 Mutual Information on First Split

Using the result from part 7, the word that produced the best initial split was 'donald'. The mutual information (a measure of 'information gain') on that split (on the training data) can be calculated by

$$I(Y, x) = H(x) - H(x, Y) = H(Y) - H(Y, x)$$

where I is the mutual information, $H(x)$ is the entropy of x, and $H(x, Y)$ is the entropy of x conditional on Y. Plugging in the values shown in Figure 2 gives

$$\begin{aligned} I(Y, x) &= H(Y) - H(Y, x) \\ &= 0.969 - \Big[P_1(1) + P_2(0.722)\Big] \\ &= 0.969 - \left[\frac{1557}{2287}(1) + \frac{730}{2287}(0.722)\right] \\ &= 0.0557 \end{aligned}$$

where $x =$ '*donald*' and Y indicates whether a headline is real or fake.

A function `mutual_info()` was written to check this result, and to calculate the mutual information for other possible splits. The code for this function is shown below; it gave a result of 0.58 for $x =$ '*donald*'.

```
def mutual_info(word, y):
    count_word = 0 #number of headlines with word in training set
    lines_with_word = []
    lines_without_word = []
    for k in range(len(training_set)):
        if word in training_set[k]:
            count_word += 1
            lines_with_word += [k]
        else:
            lines_without_word += [k]
    prob_word = count_word/len(training_set)

    y_with = np.array( [y[i] for i in lines_with_word] )
    y_without = np.array( [y[i] for i in lines_without_word] )

    prob_fake = np.count_nonzero(y)/len(y)
    H = prob_fake*np.log(prob_fake) + (1 - prob_fake)*np.log(1 - prob_fake) #
entropy before split
    H = - H/np.log(2) #convert to base 2, and apply negative

    prob_fake_with = np.count_nonzero(y_with)/len(y_with)
    Hy = prob_fake_with*np.log(prob_fake_with) + (1 - prob_fake_with)*np.log
(1 - prob_fake_with)
    Hy = - Hy/np.log(2) #entropy of headlines with word

    prob_fake_without = np.count_nonzero(y_without)/len(y_without)
    Hn = prob_fake_without*np.log(prob_fake_without) + (1 - prob_fake_without
)*np.log(1 - prob_fake_without)
    Hn = - Hn/np.log(2)  #entropy of headlines without word

    I = H - (Hy*len(y_with) + Hn*len(y_without) )/len(y)
    return I
```

8.2 Mutual Information on Later Split

The same procedure can be followed to compute the mutual information for a split on another word. Choosing a word randomly, gave a value of 0.0013 for the mutual information of a split of the training set on the word 'ways'. As expected, this value is lower than the mutual information for a split on the word 'donald' - that word was chosen as the first split precisely because it gave the most information about whether a headline was real or fake.